Clean Code Practice

In Python

This is a handbook for the clean code practice in python. This handbook covers all the best practices that should be followed while writing code to make it more readable, maintainable and extendable

Contents

Introduction

What is clean code?

Clean code is a set of rules and principle that helps to keep our code readable, maintainable, and extendable. It's one of the most important aspects of writing quality software. We spend way more time reading the code than actually writing it, which is why it's important that we write good code.

The code we write should be simple, expressive, and free from more than a few duplicates. Expressive code means that even though we're just providing instructions to a computer, it should be still be readable and clearly communicate its intent when read by humans.

Importance of clean code

- Easy to understand
- More Efficient
- Easier to maintain, scale, debug, and refactor

How the clean code should be?

- ✓ Code should be elegant and pleasing to read.
- ✓ No duplication should be allowed.
- ✓ Code should be covered with tests.
- ✓ Every function should do one thing and do it well.
- ✓ Codebase should contain only code that is needed.

PEP 8 (Python Enhancement Proposal)

PEP 8 is a style guide that describes the coding standards for Python. It's the most popular guide within the Python community.

Naming Conventions

- class names should be CamelCase (MyClass)
- variable names should be snake_case and all lowercase (first_name)
- function names should be snake_case and all lowercase (quick_sort())
- constants should be snake_case and all uppercase (PI = 3.14159)
- modules should have short, snake_case names and all lowercase (numpy)
- single quotes and double quotes are treated the same (just pick one and be consistent)

PEP 8 line formatting

- indent using 4 spaces (spaces are preferred over tabs)
- lines should not be longer than 79 characters
- avoid multiple statements on the same line
- top-level function and class definitions are surrounded with two blank lines
- method definitions inside a class are surrounded by a single blank line
- imports should be on separate lines

PEP 8 whitespace

- avoid extra spaces within brackets or braces
- avoid trailing whitespace anywhere
- always surround binary operators with a single space on either side
- if operators with different priorities are used, consider adding whitespace around the operators with the lowest priority
- don't use spaces around the = sign when used to indicate a keyword argument

PEP 8 Comments

- comments should not contradict the code
- comments should be complete sentences
- comments should have a space after the # sign with the first word capitalized
- multi-line comments used in functions (docstrings) should have a short single-line description followed by more text

Clean Code Guides

Naming Styles

Variables

- ✓ Use nouns for variable names
- ✓ Use descriptive names
- ✓ Use pronounceable names
- ✓ Avoid using ambiguous abbreviations

Example:

```
# bad practice
a = ['Cristiano Ronaldo','Lionel Messi','Sergio Ramos','Iker Cassilas']
for i in a:
    print(i)

# good practice
players = ['Cristiano Ronaldo','Lionel Messi','Sergio Ramos','Iker Cassilas']
for player in players:
    print(player)
```

Functions

- ✓ Use verbs for functions names
- ✓ Do not use different words for the same concept
- ✓ Write short and simple functions
- ✓ Functions should only perform a single task
- ✓ Keep your arguments at a minimum

Example 1:

```
# bad practice
def email():
    pass

# good practice
def send_email():
    pass
```

Example 2:

```python
# bad practice
def fetch_and_display_personnel():
    data = #...

    for person in data:
        print(data)

# good practice
def fetch_personnel():
    return #...

def display_personnel():
    for person in data:
        print(person)
```

Comments

- ✓ Don’t comment bad code, rewrite it
- ✓ Readable code doesn’t need comments
- ✓ Don’t add noise comments
- ✓ Use the correct types of comments
- ✓ Don’t leave commented out code

Example:

```python
def fetch_users(url,limit):
    '''
    fetches the users from the api and return the json data
    arguments:
    url: url to call api
    limit: number of data to be fetched
    '''
    pass
```

Avoid Duplicate Code

Duplicate code should be avoided at all costs otherwise we will end up with:

- Code that is hard to maintain or change
- Errors in code

To avoid duplication, we can use Python decorators and context managers to deal with duplicates and separate business logic from implementation details.

Example:

```python
def send_email(email_address):
    try:
        #logic related to email delivery

    except Exception as e:
        logger.exception(f'Failed to deliver message to: {e}')

    else:
        logger.info("Delivered message successfully")

def send_sms(phone_number):
    try:
        #logics related to sms delivery

    except Exception as e:
        logger.exception(f'Failed to deliver message to: {e}')

    else:
        logger.info("Delivered message successfully")
```

In the code snippet above, we have functions that are responsible for delivering email or sms messages to users as well as logging errors in case of network issues, invalid phone number or email.

We can see that we have duplicated the code for logging failures and success cases. To eliminate the duplicate code, we can refactor it using decorators.

```python
def log_message(func):
    def wrapper(*args, **kwargs):
        try:
            func(*args, **kwargs)
        except Exception as e:
            logger.exception(f'Failed to deliver message to: {e}')

        else:
            logger.info('Delivered message successfully')

    return wrapper

@log_message
def send_email(email_address):
    #logic related to email delivery

@log_message
def send_sms(phone_number):
    #logic related to sms delivery
```

Context managers can also help you reduce the amount of duplicated code when dealing with allocation and releasing resources (i.e. reading file, database connection). You can think of these as elegant solutions for separating business logic and accessing resources.

Example:

```
with open('my_file'.'w') as file:
    file.write('hello world')
```

Make Your Code SOLID

SOLID is a set of principles defined by Robert C. Martin in the ear;y 2000's and an acronym that stands for:

- ✓ Single Responsibility Principle
- ✓ Open and Closed Principle
- ✓ Lisvok Sub situation Principle
- ✓ Interface Segregation Principle
- ✓ Dependency Inversion Principle

Following those will make management of dependencies in your code easier to handle and your codebase easier to understand and extend

Single Responsibility Principle (SRP)

A class should have only one responsibility and only one reason to change i.e. the class should not perform multiple jobs.

```
#SRP violation
from os import name


class User:
    def __init__(self, name) -> None:
        self.name = name

    def save_user_to_db(self):
        db.save(self.name)

    def get_user_from_db(self):
        return db.get(self.name)
```

The class user has violated the RSP principle because it's mixed-up implementation details with business logic. Also, it has logic related to database connection and user representation.

```
#RSP implementation
class UserDB():
    def save_user_to_db(user):
        db.save(user)

    def get_user_from_db(self, name):
        return db.get(name)

class User:
    def __init__(self, name) -> None:
        self.name = name
        self._db = UserDB()

    def get_user(self):
        self._db.get_user_from_db(self.name)

    def save_user(self):
        self._db.save_user_to_db(self)
```

In order to follow the Single Responsibility principle, we can separate the User class in two classes. The first will handle database connection and the second will represent user entity in the business logic of the application.

Open/ Closed Principle (OCP)

Software entities (classes, function, module) are open for extension, but not for modification (or closed for modification). Developers often need to extend third party libraries (ex.: Django Models), but should keep the default behavior of the class we inherit from. The example below shows the implementation of the open/closed principle. Because we can't edit or modify the project source code of Django, we can make use of OCP to extend the Django Model class and add extra logic before saving any changes to the database.

```
from django.db import models

class Profile(models.Model):
    name = models.CharField(max_length = 255)
    profile_url = models.URLField(max_length = 255)

    def save(self, *args, **kwargs):
        self.check_profile_url()
        super().save(*args, **kwargs)
```

Lisvok Substitution Principle (LSP)

According to this principle every subclass that inherits from parent class, can be replaced with superclass without breaking program execution.

```
#LSP violation
class Connection:
    def connect(self, url: str):
        raise NotImplementedError

class HttpConnection(Connection):
    def connect(self, url: str, secure):
        self._connect(url, secure = secure)

class SSHConnection(Connection):
    def connect(self, url: str):
        self._connect(url)
```

Class HttpConnection violates LSP, since the contract for its parent class wasn't specified in the secured parameter. We can improve the code by adding "kwargs" parameter to HttpConnection class. This way, all subclasses of the Connection class will have the same contract for the connect method.

```python
#LSP implementation
class HttpConnection(Connection):
    def connect(self, url: str, **kwargs):
        secure = kwargs.pop('secure', True)
        self._connect(url, secure = secure)

class SSHConnection(Connection):
    def connect(self, url: str):
        self._connect(url)
```

Interface Segregation Principle (ISP)

A client should not be forced to implement an interface that it does not use

```python
#ISP violation
class Connection:
    def connect_http(self, url: str):
        raise NotImplementedError

    def connect_ssh(self, url: str):
        raise NotImplementedError

class HttpConnection(Connection):
    def connect_http(self, url):
        self._connect(url)

    def connect_ssh(self, url):
        pass

class SSHConnection(Connection):
    def connect_http(self, url):
        pass

    def connect_ssh(self, url):
        self._connect(url)
```

In the code example with classes, "HTTPConnection" and "SSHConnection" we can see an ISP violation, since those classes should implement only methods they need – not all provided by parent. Connection class and small code refactoring can ensure ISP implementation.

```
#ISP implementation
class Connection:
    def connect(self, url):
        raise NotImplementedError

class HttpConnection(Connection):
    def connect(self, url):
        self._connect_http(url)

class SSHConnection(Connection):
    def connect(self, url):
        self._connect_ssh(url)
```

Dependency Inversion Principle (DIP)

High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.

```
#DIP violation
class ConnectionService:
    def __init__(self, udp_connection) -> None:
        self.udp_connection = udp_connection

    def send(self, message):
        self.udp_connection.sendto(message, self.ip, self.port)

    def receive(self, buffer):
        while True:
            self.udp_connection.recv(buffer)
```

In the example above, we can see a DIP violation since "**ConnectionService**" class depends on "**udp connector**". Directly plugged into the initialization method, we can't extend it with "**tcp connection**" or another type of connector since we directly tied it to the udp one. Below, we can find an example with an implementation of DIP, where "**ConnectionService**" isn't directly dependent on a certain type of connector.

```
#DIP implementation
class ConnectionService:
    def __init__(self, connection) -> None:
        self._connection = connection

    def send(self, message):
        self._connection.send(message, self.ip, self.config)
    def receive(self, buffer):
        self._connection.receive(buffer)

class UDPConnection:
    def send(message, config):
        #implementation details
        pass

    def receive():
        #implementation details
        pass

class TCPConnection:
    def send(message, config):
        #implementation details
        pass

    def receive():
        #implementation details
        pass
```

Tools for Clean Code with Python

- ✓ Flake8
- ✓ Pylint
- ✓ Mypy